JVM必知必会

ranger

Published with GitBook

目錄

简介	0
JVM简介	1
内存模型	2
内存回收简介	3
内存分析命令	4
垃圾收集	5
垃圾收集器	5.1
垃圾收集算法	5.2
收集器选择	5.3
G1	5.4
jvm参数	6
类加载	7
类加载过程	7.1
类加载原理	7.2
类加载器	7.3
java class文件解析	8
jvm常用工具	9
visualvm	9.1
Memory Analyzer	9.2
案例分析	10
系统频繁Fullgc	10.1
String.intern	10.2

《JVM必知必会》

序

这里记录的是本人对JVM的总结及学习笔记，如有不足或错误请到http://www.javaranger.com留言，一经确认，立即修正

目前的内容还比较零碎，我会慢慢的做系统整理，希望这里的内容能对你产生帮助

目标读者

目前还是入门级

版权问题

如果发现本文章有版权侵犯行为，请联系我。

备注

文章内容我会一直更新，转载请注明出处，不是为了什么版权，而是因为我经常会对原文做修改，要想保证看到最新的内容，请查看原文

联系方式

- 微博：http://weibo.com/shangdiren
- 博客ranger

JAVA虚拟机简介

Java虚拟机定义

Java虚拟机有多层含义

1. 一套规范：Java虚拟机规范。定义概念上Java虚拟机的行为表现
2. 一种实现：例如HotSpot，J9，JRockit。需要实现JVM规范，但具体实现方式不需要与“概念中”的JVM一样。
3. 一个运行中的实例,某个JVM实现的某次运行的实例.
4. 只要输入为符合规范的Class文件即可执行。并非一定要执行Java程序,可以支持其它语言，像Scala、Clojure、Groovy、Fantom、Fortress、Nice、Jython、 JRuby、Rhino、Ioke、Jaskell、（C、Fortran）

JVM和JRE、JDK的关系

JVM：Java Virtual Machine，负责执行符合规范的Class文件。

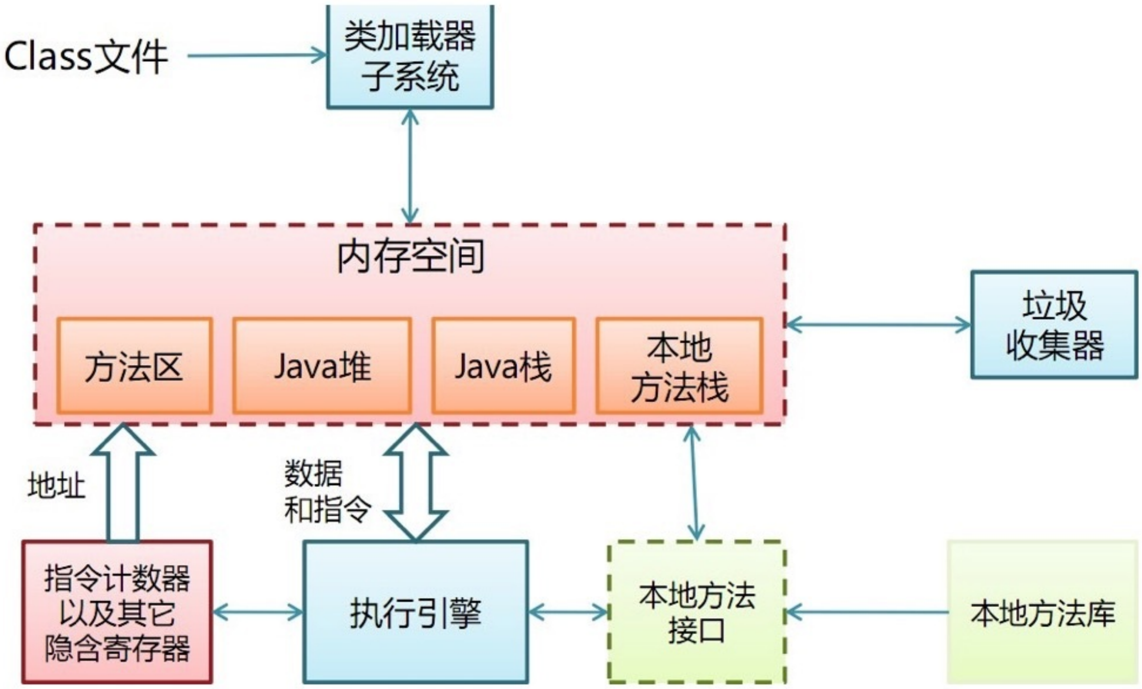
JRE：Java Runtime Environment，包含JVM和类库。

JDK：Java Development Kit，包含JRE和一些开发工具，如javac。

JVM实例和JVM执行引擎实例

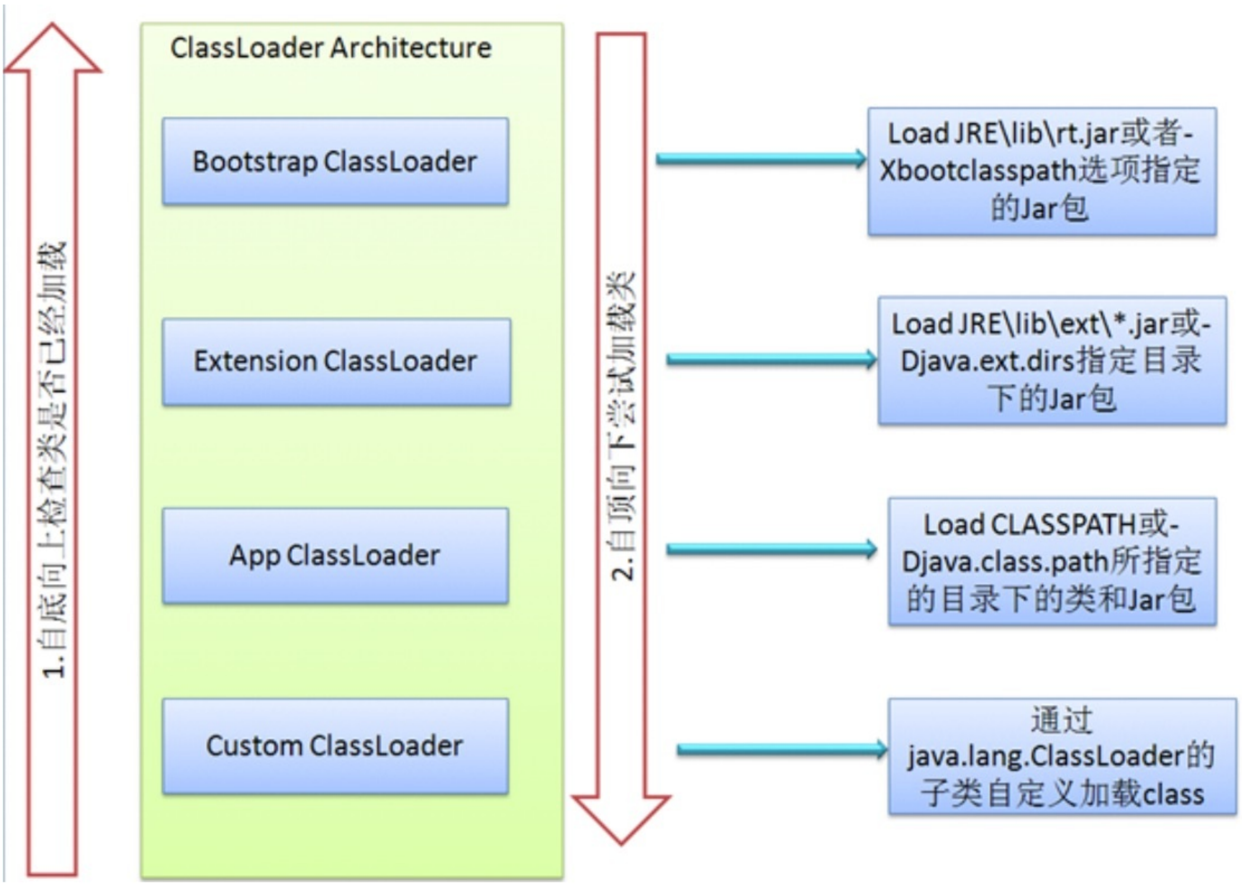
JVM实例对应了一个独立运行的java程序，而JVM执行引擎实例则对应了属于用户运行程序的线程；也就是JVM实例是进程级别，而执行引擎是线程级别的。

JVM的基本结构



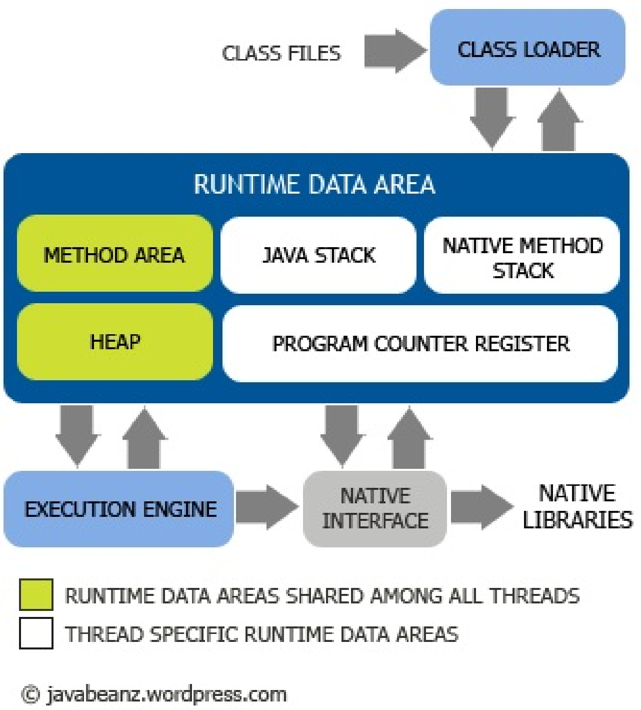
类加载子系统

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



1. Bootstrap ClassLoader负责加载$JAVA_HOME/jre/lib里所有的类库到内存，Bootstrap ClassLoader是JVM级别的，由C++实现，不是ClassLoader的子类，开发者也无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
2. Extension ClassLoader负责加载java平台中扩展功能的一些jar包，主要是由sun.misc.Launcher$ExtClassLoader实现的，是一个java类，继承自URLClassLoader超类。它将负责%JRE_HOME/lib/ext目录下的jar和class加载到内存，开发者可以直接使用该加载器。
3. App ClassLoader负责加载环境变量classpath中指定的jar包及目录中class到内存中，开发者也可以直接使用系统类加载器。
4. Custom ClassLoader属于应用程序根据自身需要自定义的ClassLoader(一般为java.lang.ClassLoader的子类)在程序运行期间，通过java.lang.ClassLoader的子类动态加载class文件，体现java动态实时类装入特性，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader。自定义ClassLoader在某些应用场景还是比较适用，特别是需要灵活地动态加载class的时候。

内存模型



这张图是我见过的最能描述JVM内存模型的图，JVM包括两个子系统和两个组件。两个子系统为：class loader（类装载）、Execution engine（执行引擎）；两个组件为：Runtime data area（运行时数据区）、Native interface(本地接口)

Class loader功能：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。程序中可以extends java.lang.ClassLoader类来实现自己的Class loader。

Execution engine功能：执行classes中的指令。任何JVM specification实现(JDK)的核心都是Execution engine，不同的JDK例如Sun的JDK和IBM的JDK好坏主要就取决于他们各自实现的Execution engine的好坏。

Native interface组件：与native libraries交互，是其它编程语言交互的接口。当调用native方法的时候，就进入了一个全新的并且不再受虚拟机限制的世界，所以也很容易出现JVM无法控制的native heap OutOfMemory。

Runtime Data Area组件：这就是我们常说的JVM的内存。主要分为五个部分： 1、Heap (堆)：一个Java虚拟实例中只存在一个堆空间 2、Method Area(方法区域)：被装载的class的信息存储在Method area的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的class文件，然后读入这个class文件内容并把它传输到虚拟机中。 3、Java Stack(java的栈)：虚拟机只会直接对Java stack执行两种操作：以帧为单位的压栈或出栈 4、Program Counter(程序计数器)：每一个线程都有它自己的PC寄存器，也是该线程启动时创建的。PC

寄存器的内容总是指向下一条将被执行指令的饿地址，这里的地址可以是一个本地指针，也可以是在方法区中相对应于该方法起始指令的偏移量。 5、Native method stack(本地方法栈)：保存native方法进入区域的地址

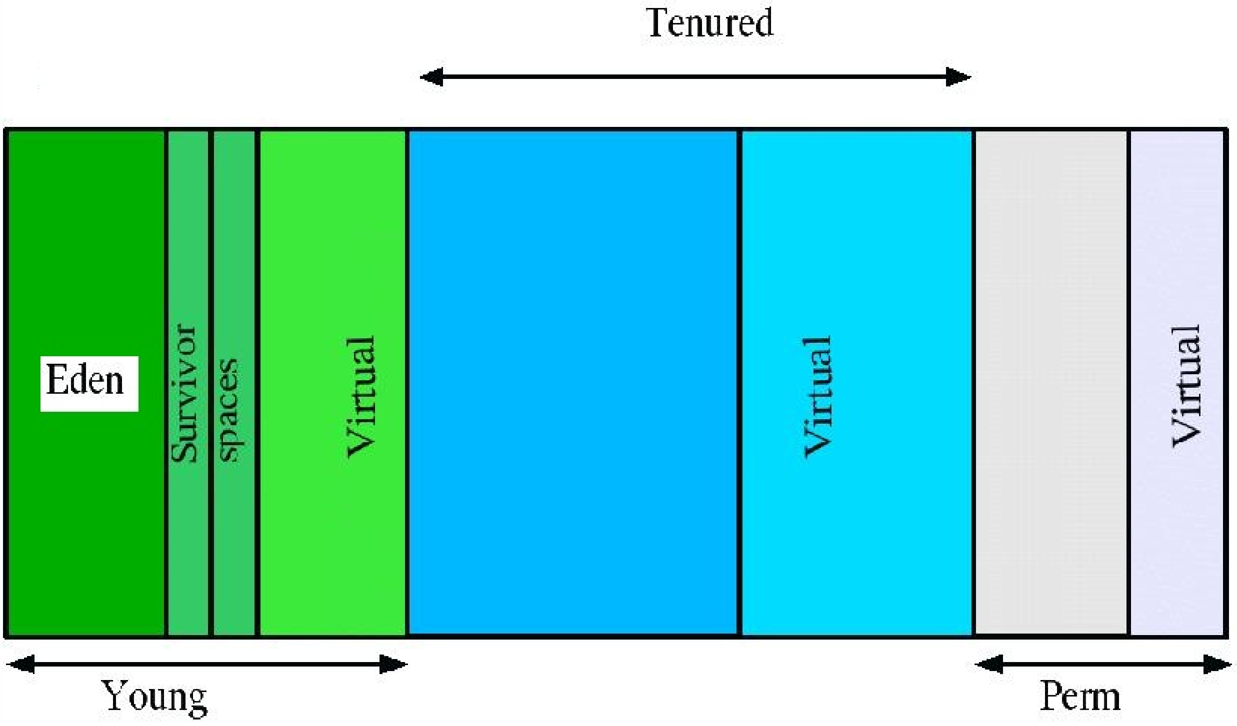
以上五部分只有Heap和Method Area是被所有线程的共享使用的；而Java stack, Program counter 和Native method stack是以线程为粒度的，每个线程独自拥有自己的部分。

（转载本站文章请注明作者和出处 JavaRanger – javaranger.com ，请勿用于任何商业用途）

本文链接: http://www.javaranger.com/archives/462

内存回收简介

Sun的JVM GC(垃圾回收)原理：把对象分为：年轻代(Young)、年老代(Tenured)、持久代(Perm)，对不同生命周期的对象使用不同的算法。(基于对对象生命周期分析)



1. Young（年轻代） 年轻代分三个区。一个Eden区，两个Survivor区。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor去也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制年老区(Tenured。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来 对象，和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor去过来的对象。而且，Survivor区总有一个是空的。
2. Tenured（年老代） 年老代存放从年轻代存活的对象。一般来说年老代存放的都是生命期较长的对象。
3. Perm（持久代） 用于存放静态文件，如今Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=进行设置。

举个例子：当在程序中生成对象时，正常对象会在年轻代中分配空间，如果是过大的对象也可能会直接在年老代生成（据观测在运行某程序时候每次会生成一个十兆的空间用收发消息，这部分内存就会直接在年老代分配）。年轻代在空间被分配完的时候就会发起内存回

收，大部分内存会被回收，一部分幸存的内存会被拷贝至Survivor的from区，经过多次回收以后如果from区内存也分配完毕，就会也发生内存回收然后将剩余的对象拷贝至to区。等到to区也满的时候，就会再次发生内存回收然后把幸存的对象拷贝至年老区。

通常我们说的JVM内存回收总是在指堆内存回收，确实只有堆中的内容是动态申请分配的，所以以上对象的年轻代和年老代都是指的JVM的Heap空间，而持久代则是之前提到的Method Area，不属于Heap。

（转载本站文章请注明作者和出处 JavaRanger – javaranger.com ，请勿用于任何商业用途）

本文链接: http://www.javaranger.com/archives/472

内存分析命令

jinfo:

```
查看Java进程的栈空间大小:sudo -u tomcat /home/java/default/bin/jinfo - ThreadStackSize 14750
查看是否使用了压缩指针:sudo -u tomcat /home/java/default/bin/jinfo -flag UseCompressedOops 14
查看系统属性:sudo -u tomcat /home/java/default/bin/jinfo -sysprops 14750
```

jstack:

```
查看一个指定的Java进程中的线程的状态:sudo -u tomcat /home/java/default/bin/jstack 14750
```

jstat:

```
查看gc的信息:sudo -u tomcat /home/java/default/bin/jstat -gcutil 14750
```

jmap&mat

```
空间中各个年龄段的空间的使用情况:sudo -u tomcat /home/java/default/bin/jmap -heap 14750
jmap指定的dump文件一定要是tomcat用户可写，比如可以新创建一个文件夹
sudo mkdir /home/memdump
sudo chown tomcat:tomcat /home/memdump
sudo -u tomcat /home/java/default/bin/jmap -dump:live,format=b,file=/home/memdump/memMap.
```

（转载本站文章请注明作者和出处 JavaRanger – javaranger.com ，请勿用于任何商业用途）

本文链接: http://www.javaranger.com/archives/1063

垃圾收集

垃圾收集器

http://www.javaranger.com/archives/636

垃圾收集器选择

JVM给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k **-XX:+UseParallelGC -XX:ParallelGCThreads=20**

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。 **-XX:ParallelGCThreads=20**：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。 java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 **-XX:+UseParallelOldGC**

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC **-XX:MaxGCPauseMillis=100**

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 **-XX:+UseAdaptiveSizePolicy**

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 **-XX:+UseConcMarkSweepGC -XX:+UseParNewGC**

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC **-XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection**

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。 **-XX:+UseCMSCompactAtFullCollection**：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

（转载本站文章请注明作者和出处 JavaRanger – javaranger.com， 请勿用于任何商业用途）

本文链接: http://www.javaranger.com/archives/1832

Garbage First介绍

Garbage First简称G1，它的目标是要做到尽量减少GC所导致的应用暂停的时间，让应用达到准实时的效果，同时保持JVM堆空间的利用率，其最大的特色在于允许指定在某个时间段内GC所导致的应用暂停的时间最大为多少，例如在100秒内最多允许GC导致的应用暂停时间为1秒，这个特性对于准实时响应的系统而言非常的吸引人，这样就再也不用担心系统突然会暂停个两三秒了。

目标

从设计目标看G1完全是为了大型应用而准备的。

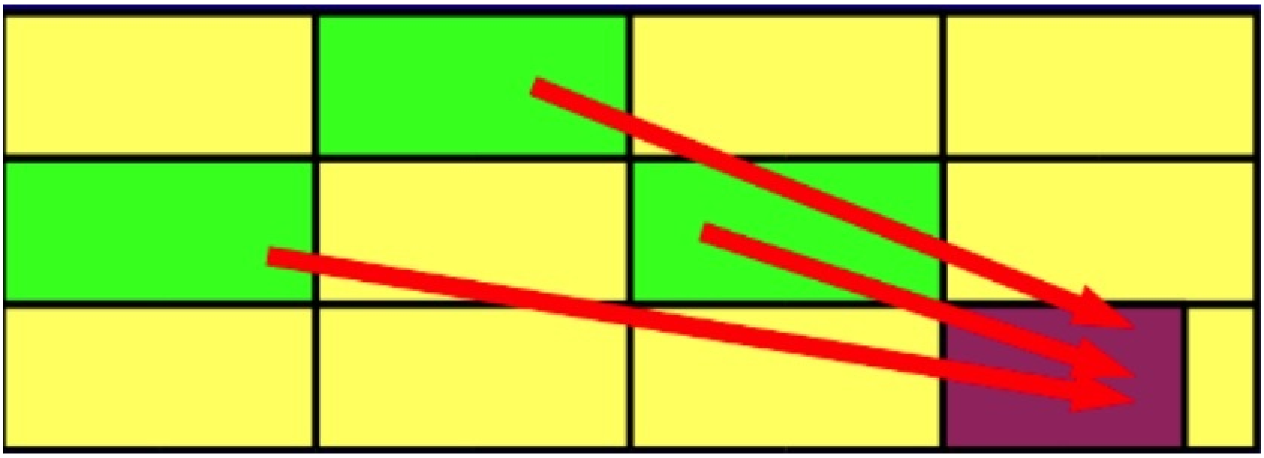
支持很大的堆

高吞吐量

- 支持多CPU和垃圾回收线程
- 在主线程暂停的情况下，使用并行收集
- 在主线程运行的情况下，使用并发收集

实时目标：可配置在N毫秒内最多只占用M毫秒的时间进行垃圾回收 当然G1要达到实时性的要求，相对传统的分代回收算法，在性能上会有一些损失。

算法详解



G1可谓博采众家之长，力求到达一种完美。他吸取了增量收集优点，把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；同时，他也吸取了CMS的特点，把这个垃圾回收过程分为几个阶段，分散一个垃圾回收过程；而且，G1也认同分代垃圾回收的思想，认为不同对象的生命周期不同，可以采取不同收集方式，因此，它也支持分

代的垃圾回收。为了达到对回收时间的可预计性，G1在扫描了region以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的region，以便快速回收空间（要复制的活跃对象少了），因为活跃对象小，里面可以认为多数都是垃圾，所以这种方式被称为Garbage First（G1）的垃圾回收算法，即：垃圾优先的回收。

回收步骤：

1. 初始标记（**Initial Marking**）

G1对于每个region都保存了两个标识用的bitmap，一个为previous marking bitmap，一个为next marking bitmap，bitmap中包含了一个bit的地址信息来指向对象的起始点。

开始Initial Marking之前，首先并发的清空next marking bitmap，然后停止所有应用线程，并扫描标识出每个region中root可直接访问到的对象，将region中top的值放入next top at mark start（TAMS）中，之后恢复所有应用线程。

触发这个步骤执行的条件为：

G1定义了一个JVM Heap大小的百分比的阀值，称为h，另外还有一个H，H的值为(1-h)*Heap Size*，目前这个*h*的值是固定的，后续*G1*也许会将其改为动态的，根据*jvm*的运行情况来动态的调整，在分代方式下，*G1*还定义了一个*u*以及*soft limit*，*soft limit*的值为*H-u*Heap Size，当Heap中使用的内存超过了soft limit值时，就会在一次clean up执行完毕后在应用允许的GC暂停时间范围内尽快的执行此步骤；

在pure方式下，G1将marking与clean up组成一个环，以便clean up能充分的使用marking的信息，当clean up开始回收时，首先回收能够带来最多内存空间的regions，当经过多次的clean up，回收到没多少空间的regions时，G1重新初始化一个新的marking与clean up构成的环。

2.并发标记（**Concurrent Marking**）

按照之前Initial Marking扫描到的对象进行遍历，以识别这些对象的下层对象的活跃状态，对于在此期间应用线程并发修改的对象的以来关系则记录到remembered set logs中，新创建的对象则放入比top值更高的地址区间中，这些新创建的对象默认状态即为活跃的，同时修改top值。

3.最终标记暂停（**Final Marking Pause**）

当应用线程的remembered set logs未满时，是不会放入filled RS buffers中的，在这样的情况下，这些remebered set logs中记录的card的修改就会被更新了，因此需要这一步，这一步要做的就是把应用线程中存在的remembered set logs的内容进行处理，并相应的修改remembered sets，这一步需要暂停应用，并行的运行。

4.存活对象计算及清除（**Live Data Counting and Cleanup**）

值得注意的是，在G1中，并不是说Final Marking Pause执行完了，就肯定执行Cleanup这步的，由于这步需要暂停应用，G1为了能够达到准实时的要求，需要根据用户指定的最大的GC造成的暂停时间来合理的规划什么时候执行Cleanup，另外还有几种情况也是会触发这个步骤的执行的：

G1采用的是复制方法来进行收集，必须保证每次的”to space”的空间都是够的，因此G1采取的策略是当已经使用的内存空间达到了H时，就执行Cleanup这个步骤；

对于full-young和partially-young的分代模式的G1而言，则还有情况会触发Cleanup的执行，full-young模式下，G1根据应用可接受的暂停时间、回收young regions需要消耗的时间来估算出一个yound regions的数量值，当JVM中分配对象的young regions的数量达到此值时，Cleanup就会执行；partially-young模式下，则会尽量频繁的在应用可接受的暂停时间范围内执行Cleanup，并最大限度的去执行non-young regions的Cleanup。

JVM参数

http://www.javaranger.com/archives/382

类加载器

```
public class ClassLoaderTest {
    public static void main(String[] args) {
        /**
         * 我们无法获得引导类加载器，因为它是使用c实现的，而且使用引导类加载器加载的类通过getClassLoad
         */
        List<URL> list = Arrays.asList(sun.misc.Launcher.getBootstrapClassPath().getURLs(
        for(URL url : list){
            System.out.println(url.toString());
        }
    }
}
```

Output：

```
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/resources.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/rt.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/sunrsasign.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/jsse.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/jce.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/lib/charsets.jar
file:/D:/Program%20Files/Java/jdk1.6.0_13/jre/classes
```

从这个例子中我们可以看出Bootstrap ClassLoader加载的为$JAVA_HOME/jre/lib目录下的jar包。

Bootstrap ClassLoader、Extension ClassLoader、App ClassLoader三者的关系如下：Bootstrap ClassLoader由JVM启动，然后初始化sun.misc.Launcher，sun.misc.Launcher初始化Extension ClassLoader、App ClassLoader。Bootstrap ClassLoader是Extension ClassLoader的parent，Extension ClassLoader是App ClassLoader的parent。但是这并不是继承关系，只是语义上的定义，基本上，每一个ClassLoader实现，都有一个Parent ClassLoader。可以通过ClassLoader的getParent方法得到当前ClassLoader的parent。Bootstrap ClassLoader比较特殊，因为它不是java class所以Extension ClassLoader的getParent方法返回的是NULL。我们举下面的实例说明一下：

```
public class ClassLoaderTest2 {
     public static void main(String[] args) {
ClassLoader loader = Thread.currentThread().getContextClassLoader();
System.out.println("current loader---->"+loader);
     System.out.println("parent loader-->"+loader.getParent());
System.out.println("grandparent loader->"+loader.getParent().getParent());
             }
}
```

Output：

```
current loader---->sun.misc.Launcher$AppClassLoader@19821f
parent loader---->sun.misc.Launcher$ExtClassLoader@addbf1
grandparent loader---->null
```

了解了ClassLoader的原理和流程以后，我们可以试试自定义ClassLoader。关于自定义ClassLoader： 由于一些特殊的需求，我们可能需要定制ClassLoader的加载行为，这时候就需要自定义ClassLoader了.

自定义ClassLoader需要继承ClassLoader抽象类，重写findClass方法，这个方法定义了ClassLoader查找class的方式。

主要可以扩展的方法有：

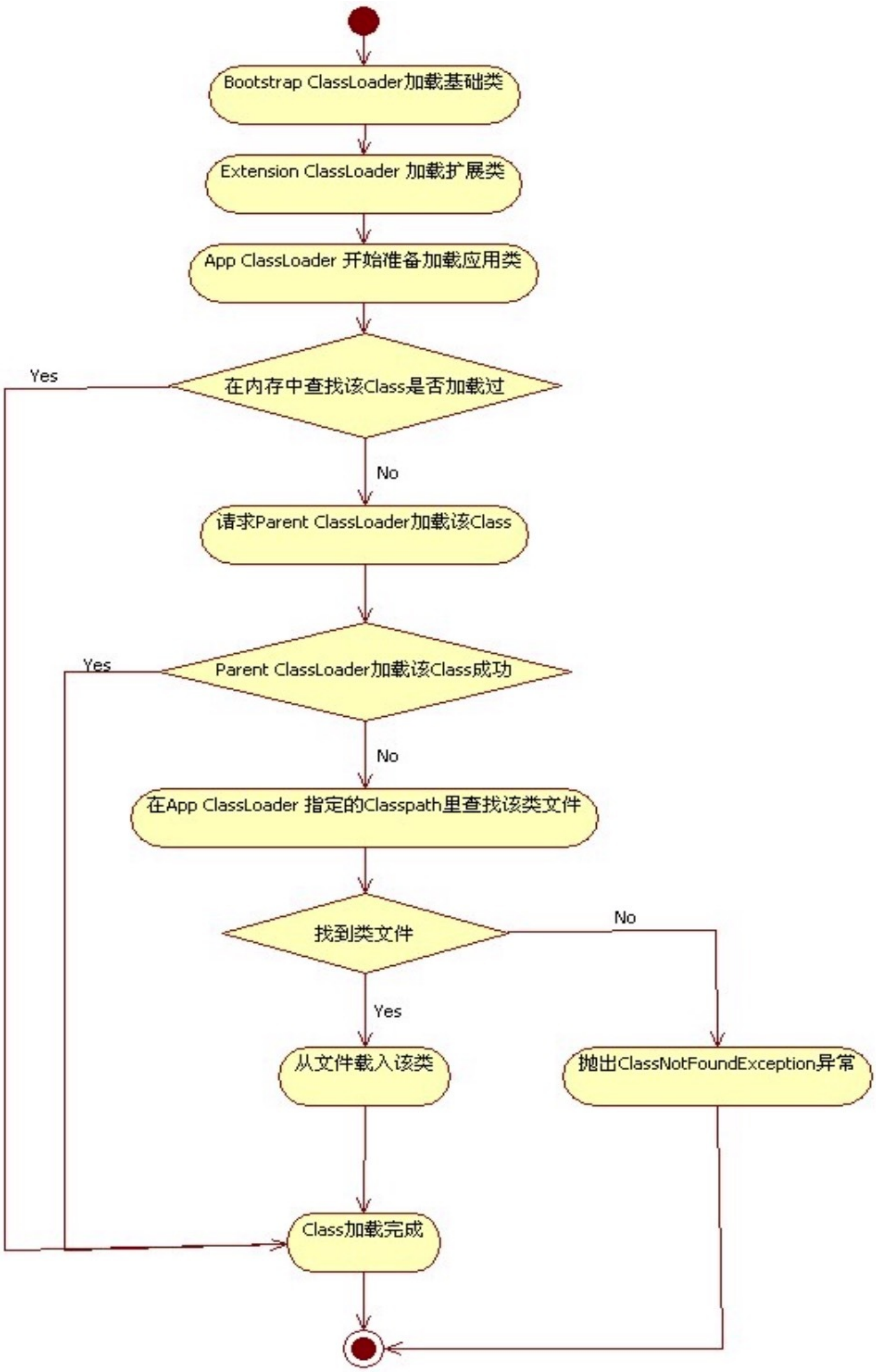
- findClass 定义查找Class的方式
- defineClass 将类文件字节码加载为jvm中的class
- findResource 定义查找资源的方式

如果嫌麻烦的话，我们可以直接使用或继承已有的ClassLoader实现，比如 java.net.URLClassLoader java.security.SecureClassLoader java.rmi.server.RMIClassLoader sun.applet.AppletClassLoader Extension ClassLoader 和 App ClassLoader都是 java.net.URLClassLoader的子类。 这个是URLClassLoader的构造方法： public URLClassLoader(URL[] urls, ClassLoader parent) public URLClassLoader(URL[] urls) urls参数是需要加载的ClassPath url数组，可以指定parent ClassLoader，不指定的话默认以当前调用类的ClassLoader为parent。下面以一个例子加以说明： Java代码1：

```
public class ClassWillBeLoaded {//这个类是要被装载的一个类(测试类).
    public static void main(String[] args) {
        ClassWillBeLoaded obj = new ClassWillBeLoaded();
    }
    public String doTask(String str1,String str2){
        return str1+" "+str2;
    }
```

```
public class ClassLoaderTest3 {//使用反射机制调用通过URLClassLoader装载的类中的doTask方法
    public static void main(String[] args) throws MalformedURLException, ClassNotFoundExc
        URL url = new URL("file:D:/share/ClassLoadTest.jar");
        URL[] urls = {url};
        ClassLoader classLoader = new URLClassLoader(urls);
    Thread.currentThread().setContextClassLoader(classLoader);//设置该线程的上下文ClassLoade
        Class clazz=classLoader.loadClass("classLoader.ClassWillBeLoaded");//使用loadClass
Method taskMethod = clazz.getMethod("doTask", String.class, String.class);//然后我们就可以用
Object returnValue = taskMethod.invoke(clazz.newInstance(),"test","success");
        System.out.println((String)returnValue);
    }   }
```

要哪一个class loader加载呢？答案在于全盘负责委托机制，这是出于安全的原因。每次只要一个class被loaded，系统的class loader就首先被调用。然而它不会立即去load这个这个类。取而代之的是，他会把这个task委托给他的parent class loader，也就是extension class loader;同样的，extension class loader也会委托给它的parent class loader也就是bootstrap class loader。因此，bootstrap class loader总是被给第一个去load class的机会。如果bootstrap class loader找不到类的话，那么extension class loader将会load，如果extension class loader也没有找到对应的类的话，system class loader将会执行这个task，如果system class loader也没有找到的话，java.lang.ClassNotFoundException将会被抛出。另外一个原因是避免了重复加载类，每一次都是从底向上检查类是否已经被加载，然后从顶向下加载类，保证每一个类只被加载一次。


Bootstrap ClassLoader加载基础类
Extension ClassLoader 加载扩展类
App ClassLoader 开始准备加载应用类
在内存中查找该Class是否加载过
Yes
No
请求Parent ClassLoader加载该Class
Parent ClassLoader加载该Class成功
Yes
No
在App ClassLoader 指定的Classpath里查找该类文件
找到类文件
No
Yes
从文件载入该类
抛出ClassNotFoundException异常
Class加载完成

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只被所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

Java Class文件格式解析

http://www.javaranger.com/archives/1875

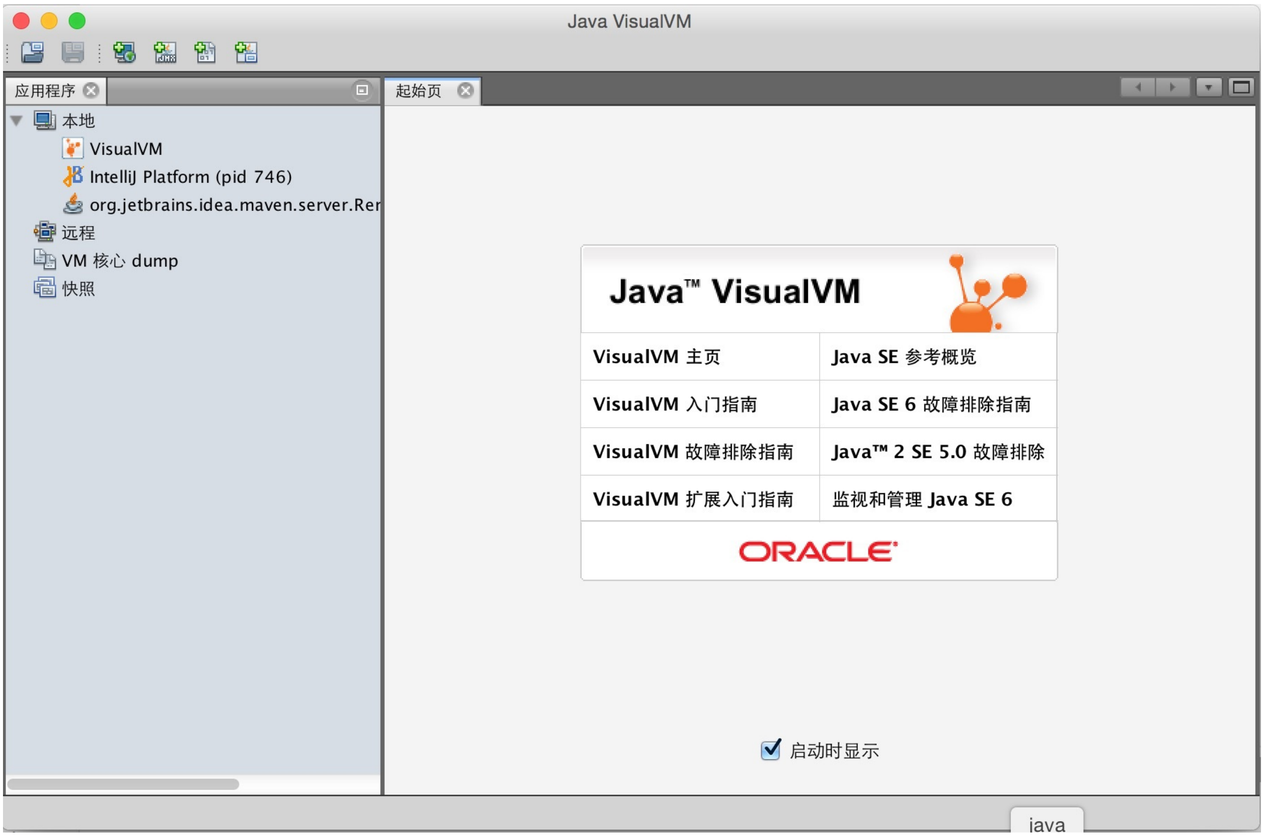
VisualVM

简介

VisualVM 提供在运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中可以方便、快捷地查看多个 Java 应用程序的相关信息。因为JDK自带该工具，且属于免费软件，我们对java应用程序进行简单的监控分析时直接用该工具，当然如果有更复杂、更专业的监控分析需求，则最好选择商用软件。

使用

JDK1.6u7以后版本已携带该工具，如果你安装的JDK并未携带该工具，读者可从 https://visualvm.java.net 下载，直接终端中输入jvisualvm回车或进入jdk的bin目录后输入命令即可打开，打开界面如下所示：



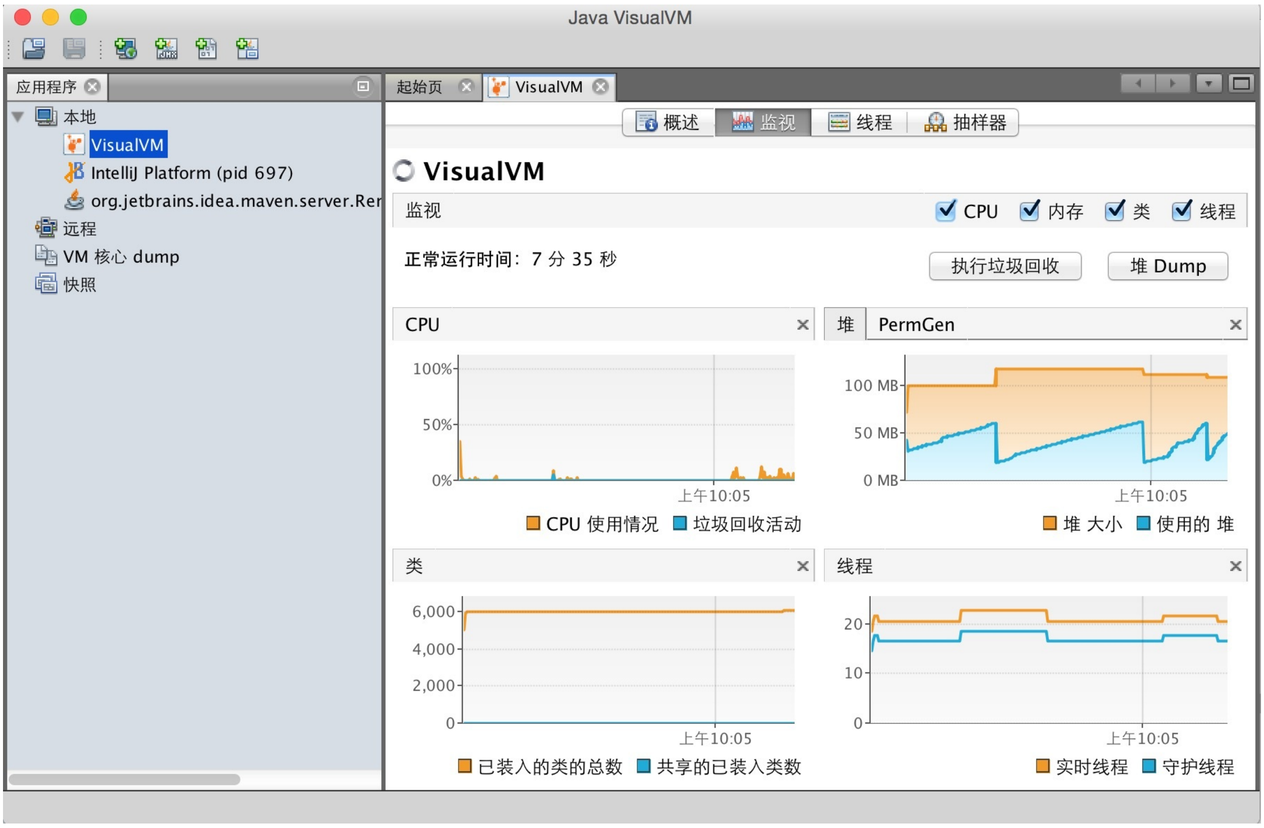
在左侧的“应用程序”窗口中，可以快速查看本地和远程 JVM 上运行的 Java 应用程序。接下来我们比照命令行工具简单操作一下。

1. 可查看JVM进程及进程配置、环境等相关信息，功能类jps、jinfo命令行。 这里我们就看viaualvm的一些信息，应用程序-本地栏下出现一个ViaualVM的节点，我们双击这个节点，进入以下界面：

从该界面概述选项卡里我们可以查看进程pid、JVM参数等信息。

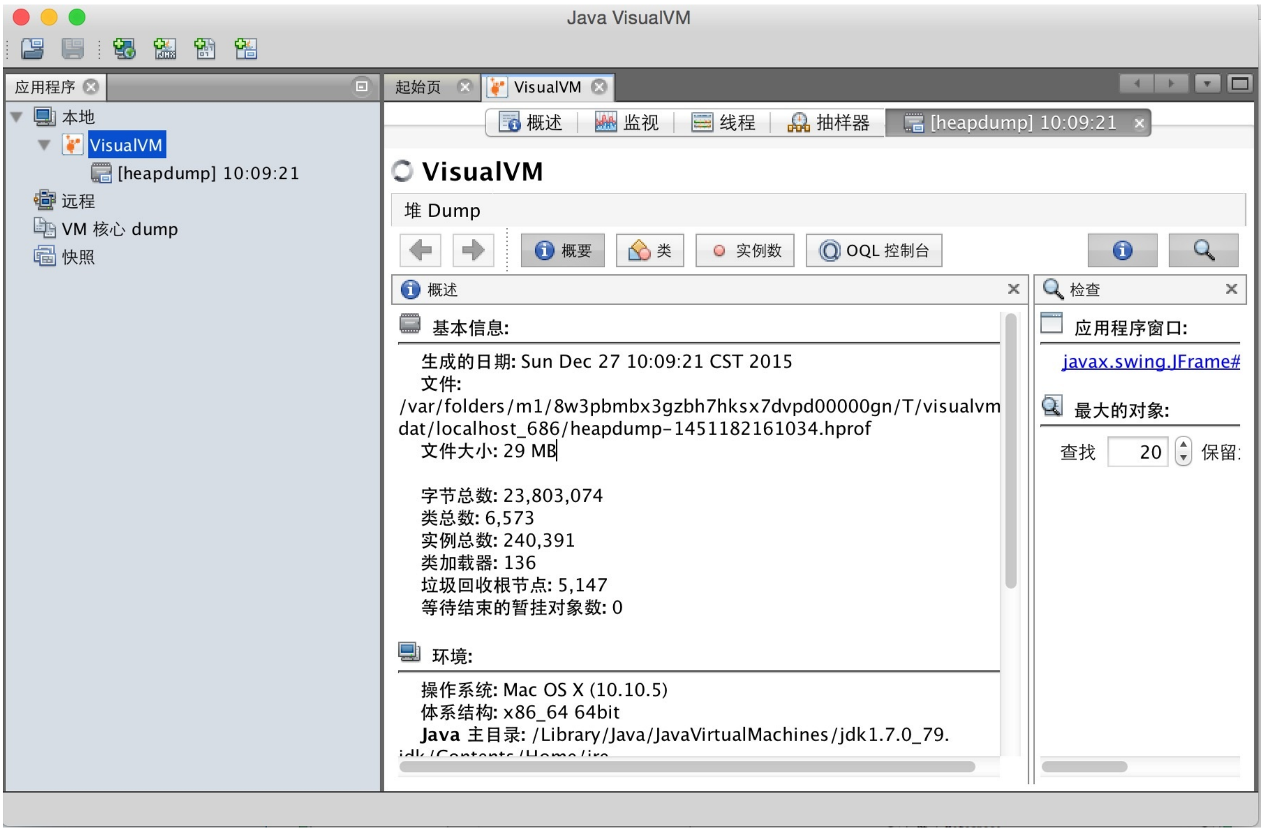
2. 可查看应用程序内存、CPU、堆、方法区、线程等信息，功能类jstat、jstack。

我们点击“监视”、“线程”选项卡可直观地查看CPU、内存、类、线程、垃圾回收情况等信息。如下图所示：



1. 生成Dump、分析Dump、生成快照等，功能类jmap、jhat。

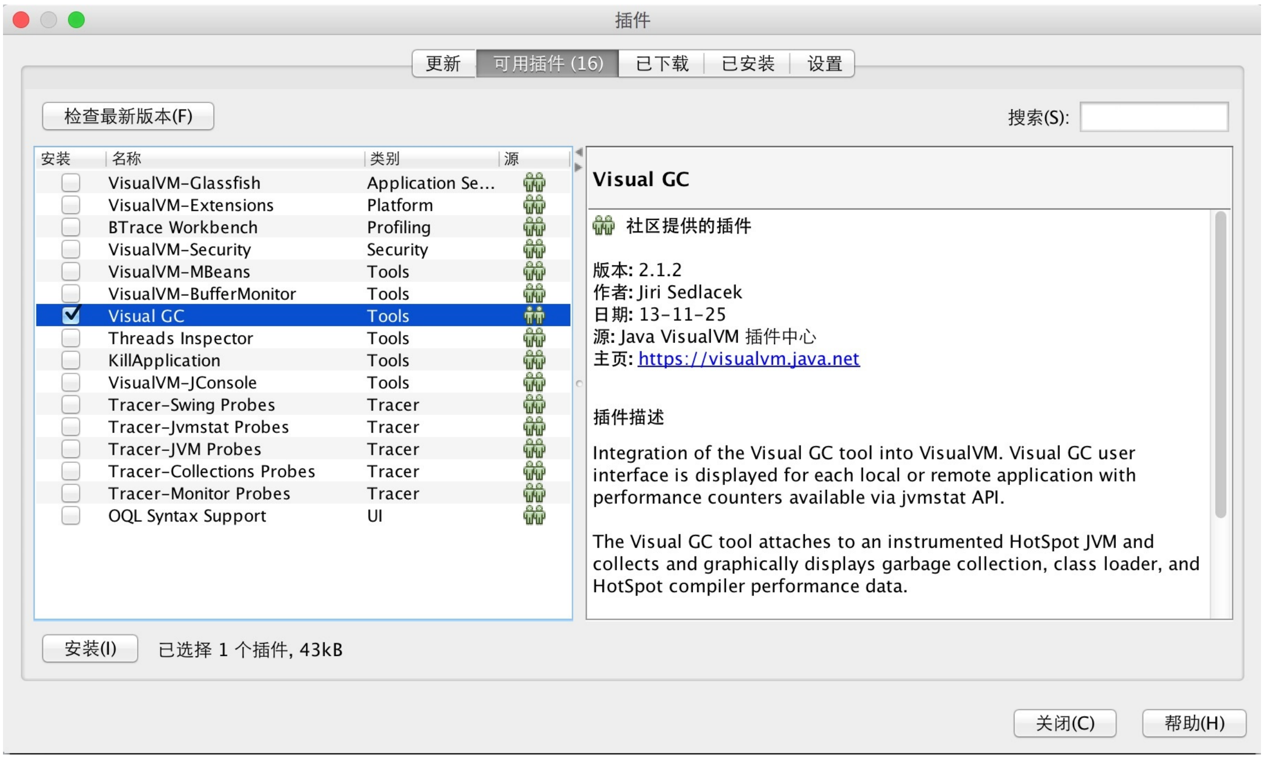
右键单击应用程序节点将打开弹出式菜单，从该弹出式菜单中可以生成线程 dump 或堆 dump。生成dump将扩展到应用程序节点下，如下两张图所示：



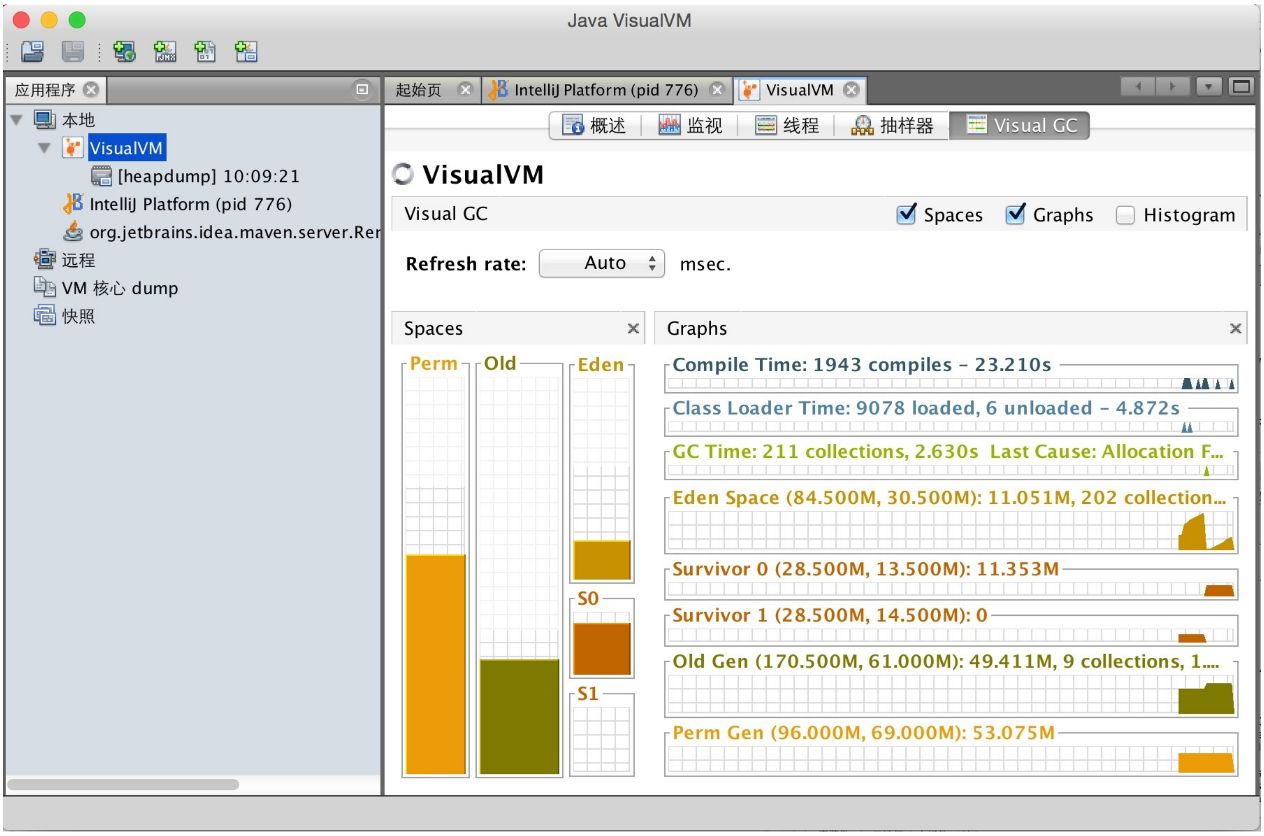
1. 其它功能及功能扩展。

我们可以在应用程序的Profiler选项卡下对cup和内存的性能进行分析。

VisualVM还可以很方便地扩展功能，大家可以点击工具菜单，进入插件界面，点击可用插件，然后就可以对其功能进行扩展了，如下图所示：



这里我们下载安装Visual GC。下载安装成功后重新打开应用程序节点VisualVM，我们可以看到界面中多了一个Visual GC的选项卡，打开后入下图所示：



案例分析

一些案例

系统频繁Full gc问题分析及解决办法

http://www.javaranger.com/archives/1627

Java6,7,8中的String.intern() – 字符串常量池

http://www.javaranger.com/archives/1852